Rapid Application of Lightweight Formal Methods for Consistency Analyses

Martin S. Feather

Abstract—Lightweight formal methods promise to yield modest analysis results in an extremely rapid manner. To fulfill this promise, they must be able to work with existing information sources, be able to analyze for manifestly desirable properties, be highly automated (especially if dealing with voluminous amounts of information), and be readily customizable and flexible in the face of emerging needs and understanding. Two pilot studies investigate the feasibility of lightweight formal methods that employ a database as the underlying reasoning engine to perform the analyses. The first study concerns aspects of software module interfaces, the second test logs' adherence to required and expected conditions.

Index Terms—Consistency checking, formal methods, interface checking, test-log checking, database-based analysis, NASA.

1 Introduction

Critical software systems often warrant high levels of assurance as to the correctness of their design and implementation. Increasingly, formal methods are being applied in conjunction with traditional testing as a means to achieve these high levels of assurance. In such a context, formal methods are just another analysis technique, and the choice of when and where to apply them should be justified in terms of their cost-effectiveness. Criteria that enter into this determination include the ease of application of the methods, and the timeliness and value of their results.

Jackson and Wing [8] in their contribution to a roundtable discussion use the term *lightweight formal methods* to refer to formal methods intended to be particularly amenable to rapid application, and thus have the capacity to yield results in a cost-effective and timely fashion. Traditionally, use of tool-based formal methods in the arena of software validation and verification has applied theorem proving to confirm properties of formal specifications. Theorem proving can indeed be applied to conduct deep and significant analyses, but often requires a large investment of effort to prepare for its application. In contrast, lightweight formal methods occupy a different place in the spectrum of analysis techniques. They have more modest analysis goals, and employ tools that require less preparatory work to apply.

The goal of rapid application ensures that the analysis results become available early in the development process. This has the obvious benefit that the developers become alerted to discovered problems early rather than late in the development process, so saving them the considerable effort of fixing errors downstream [2]. Thus, the analysis results are both timely and potentially valuable. Furthermore, the need to achieve rapidity of analysis has the side effect of constraining the analysis method to be one which is easy and simple to apply (anything otherwise would fail to be sufficiently rapid!). That is, by their nature, rapidly applied analysis methods are inherently inexpensive.

The approach we have followed employs a database as the underlying reasoning engine to perform analyses. The rationale for this choice is described in Section 2, along with the background to the two pilot studies used to investigate the feasibility of this approach. The first pilot study is described in Section 3, along with the intermediate conclusions drawn from that effort. The second pilot study employed the same tool in support of analysis, and was applied to a different aspect of that same project; it is described in Section 4. An overall discussion concludes the paper in Section 5.

2 Approach

Meeting the goal of rapid application necessitates a judicious simultaneous choice of analysis objective and analysis method. It must be relatively easy and speedy to:

1) acquire the information to be analyzed in the form required for analysis,
2) decide what to analyze this information for,
3) actually perform the analysis itself, and
4) interpret the results of the method.

Together, 1) and 2) imply a need to employ analysis techniques that work with available sources of information and to analyze for properties that are readily seen to be required and readily expressed to the analysis tool. Numbers 3) and 4) imply a need to apply highly automated analysis methods that are both rapid and flexible.

Our approach is distinguished from previous approaches (both lightweight and heavyweight) by the fact that we frame the analysis problem in terms of database queries, and we use a database as the underlying analysis engine. The information to be analyzed is loaded as data into the database, and the properties to be analyzed for are cast as database queries. The database itself evaluates those queries, and the query results provide the detailed analysis results.

• *M.S. Feather is with the Jet Propulsion Laboratory, California Institute of Technology, MS 125-233, 4800 Oak Grove Dr., Pasadena CA 91109. E-mail: martin.s.feather@jpl.nasa.gov.*

Manuscript received 15 Sept. 1997; revised 13 Mar. 1998.
Recommended for acceptance by B. Nuseibeh and C. Ghezzi.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 107207.

The strengths and weaknesses of a database used as an analysis engine differ somewhat from those of reasoning engines typically brought to bear for software analysis. A typical database provides a flexible query language, user-definable schema with which to express relationships among data, support for loading data into the database, and powerful report generation capabilities. These characteristics support the four criteria identified above. The user-definable schema and support for data entry facilitate working with available sources of information in a large variety of formats. A flexible query language permits the easy expression of a wide range of properties to be analyzed for. The underlying query optimization mechanism relieves the user from the burden of (re)expressing the property (query) just so as to achieve efficiency of analysis. The ability to refine and compose queries provides flexibility to evolve the analysis. Result reporting and categorization supports ease of expressing results (e.g., listing anomalies and their details).

Of course, a database is not suited to every kind of analysis. The most important constraining factor is the need to work with *explicit* information (so that it can be loaded as data into the database). In contrast, other analysis techniques are commonly based upon mechanisms that work with implicit forms of expression. For example, state-exploration techniques may work with a program-like description, implying a state space; they generate and explore this implicit state space themselves. The ability to analyze properties of implicit forms of information is highly desirable—it may yield results that are difficult for humans to reliably ascertain by manual inspection. Theorem proving can be applied to analyze properties of implicit descriptions of information that, if made explicit, would be infinite in size. Yet working with implicit information often necessitates a good match of both notation and scale between the form of the information to be analyzed, and the form and scale of input that the analysis tool can accept and tractably reason with. Mismatches in either form or scale can be bridged, but typically only with a considerable investment of time and effort.

Another distinguishing characteristic of our approach is the emphasis on working with available information. Our motivation is twofold: first and foremost, to achieve the rapidity of analysis that we desire; second, to be able to ascribe the benefits of the approach to mechanized analysis rather than to the involvement and insight of a skilled analyst. This last point is important if we seek to transfer the technique into widespread use, since analysis technology is readily replicated, whereas skilled analysts are not.

This paper reports on two closely linked pilot studies that rapidly apply lightweight formal methods employing a database as reasoning engine. These pilot studies investigate feasibility by application to real problems, but do not replace any of the inspection and testing activities that the spacecraft developers must currently perform. The intent is that a pilot study will indicate whether a technique has promise, and if so, indicate how it should be put to practical use in future projects.

The area of the studies is the ongoing design and development of spacecraft software. The particular spacecraft we have studied is NASA's New Millennium project's Deep Space-1, in particular, the Autonomy software intended to control that spacecraft. The spacecraft project adopted a fast-paced rapid prototyping style of development, and employed relatively complex on-board software. Rapidly applied analysis methods were thus highly appropriate. Our focus was on two aspects of this project: analyzing for consistency and completeness properties of interfaces between some of the software modules, and analyzing transcripts generated during testing for adherence to some of the requirements. Analyses such as these are likely to be useful in a wide range of software systems, not just spacecraft software.

3 Pilot Study I—Analyzing Software Interfaces

The software system that controls the spacecraft is subdivided into several major modules, which communicate via message passing. Separate teams of developers are responsible for the design and development of each of these modules. A previous study of safety-critical, embedded systems [11] identified interfaces as a major source of software errors. This suggested that the interfaces between this spacecraft's modules would be an area ripe for analysis. Furthermore, lightweight analysis was anticipated to be applicable to reveal discrepancies between different modules at the interface level.

The four requirements for rapid analysis led to the following decisions:

1) *To rapidly acquire the information to be analyzed.* In the early stages of development, the interface of each module was documented diagrammatically to facilitate coordination and understanding between the development teams. Thus, the analysis process was targeted to work with this same information source.
2) *To rapidly decide what to analyze this information for:* Manifestly desirable properties of consistency and completeness were easy to postulate. Additional information present in the diagrams, in the form of simple causality information, also served as a source of further obvious analysis opportunities.
3) *To rapidly perform the analysis itself.* A database with a powerful and flexible query mechanism was chosen to serve as the analysis tool. The expectation was that it would be straightforward to design a database schema customized for holding the information content implied by the diagrams, load the diagrams' information into that database, and express the analyses as database queries. Thereafter, the query mechanism of the database itself would perform the analyses rapidly and automatically.
4) *To rapidly interpret the results of the method.* Discrepancies that this analysis revealed would be readily traceable to the interface diagrams, and so would be easy to interpret. The exercise showed that some iterative refinement of the queries was needed to separate those discrepancies attributable to obviously missing information from those discrepancies that were more indicative of unintended omissions, etc.

Again, the flexibility and simplicity of a database analysis engine proved the key to the rapidity of these iterative refinements.

3.1 Available Information

In the early stages of development, the interface of each module was documented diagrammatically to facilitate coordination and understanding between the development teams.

Fig. 1 shows an example of one of these diagrams. The software module is drawn as a rectangular box; arrows entering into and emanating out from this box indicate the types of messages that can be received by and sent from this module.

In more detail:

- All the possible input message types of the module are shown as incoming arrows on the left.
- All the possible output message types of the module are shown as outgoing arrows on the right.
- The name of the message type is shown in capital letters above each arrow, along with the message parameters (if any) inside angled brackets "<...>".
- For each *incoming* arrow, the names of the modules from which messages of that type may *originate* are shown below the arrow.
- For each *outgoing* arrow, the names of the modules to which messages of that type are *sent* are shown below the arrow.
- Cause-effect relationships between message types are shown as dotted lines going across the inside of the box:
  - A dotted line going from an incoming message type to an outgoing message type indicates that receipt of such an incoming message *may* lead to the software module producing such an outgoing message. We will refer to such dotted lines as denoting *"explicit"* cause-effect links.
  - A dotted line going from an outgoing message type to an incoming message type indicates that sending of such an outgoing message may (via the actions of *other* software modules) lead to the receipt of such an incoming message. We will refer to such dotted lines as denoting *"implicit"* cause-effect links.

3.2 Analysis Objectives

The objectives set for analysis were to look for instances of the following potential problems within the set of software interface diagrams:

- *"Dangling" outgoing message type.* A message type on an outgoing arrow of module M1 listed as going to some module M2, but *not* listed on module M2's diagram as an incoming message type from module M1.
- *"Dangling" incoming message type.* A message type on an incoming arrow of module M1 listed as coming from some module M2, but *not* listed on module M2's diagram as an outgoing message type to module M1.
- *"Mismatched" parameters.* A message type whose list of parameters in one module is not identical to its list of parameters in some other module.
- *"Miraculous" implicit cause-effect link.* An implicit cause-effect link (i.e., a link from an outgoing message type (T1, say), to an incoming message type (T2, say), such that there does *not* exist a chain of explicit cause-effect links and correspondences between outgoing and incoming messages that connects T1 to T2. See Fig. 2 for an example of an implicit cause effect link for which the corresponding chain exists; had any one of the elements of that chain been absent (e.g., the explicit cause-effect link in M2), then the chain would have been broken, and M1's implicit link would have been deemed "miraculous."
- *"Omitted" implicit cause-effect link.* Omission of an implicit cause-effect link (from outgoing message type T1 to incoming essage type T2) for which there *does* exist a chain of explicit cause-effect links that connect T1 to T2.

3.3 Analysis Process

Selection of the analysis tool was driven by the following considerations:

- pressing need for rapidity of analysis results,
- potential need to scale to voluminous quantity of data, and
- relatively straightforward nature of analysis calculations.

Together, these motivated the selection of a *database* with a powerful and flexible query mechanism to serve as the analysis tool. A powerful and flexible database query mechanism would enable rapid analysis. The database itself would easily handle voluminous amounts of data, while automatic query optimization would ensure efficient analysis. Finally, since the analysis calculations were expected to be relatively straightforward, it was anticipated that the simple reasoning capabilities of a database would suffice. Had the analysis required, say, reasoning about symbolic expressions with arithmetic inequalities, then it is likely that a more sophisticated tool such as a theorem prover would have been needed.

Our choice was to use AP5 [3], a research-quality advanced database tool developed at the University of Southern California. AP5 provides an entity-relationship-like data-model convenient for representation, facilitating the design of the data schema to hold the information. AP5 exists as an extension of Common Lisp, itself a powerful and convenient programming language environment, suited to the ad hoc programming needed for the process of data-entry. AP5 provides a powerful definition capability and query interface, facilitating the expression of analyses as database queries, and has a sophisticated underlying query optimization mechanism.

Having selected the AP5 database system as the analysis tool, the analyses were achieved by: 1) preparing a database representation to hold the information content of the software interface diagrams, 2) loading the information into the database, 3) issuing the appropriate database queries corresponding to each of the problem category analyses, and 4) interpreting the results.

These steps are now discussed in detail.

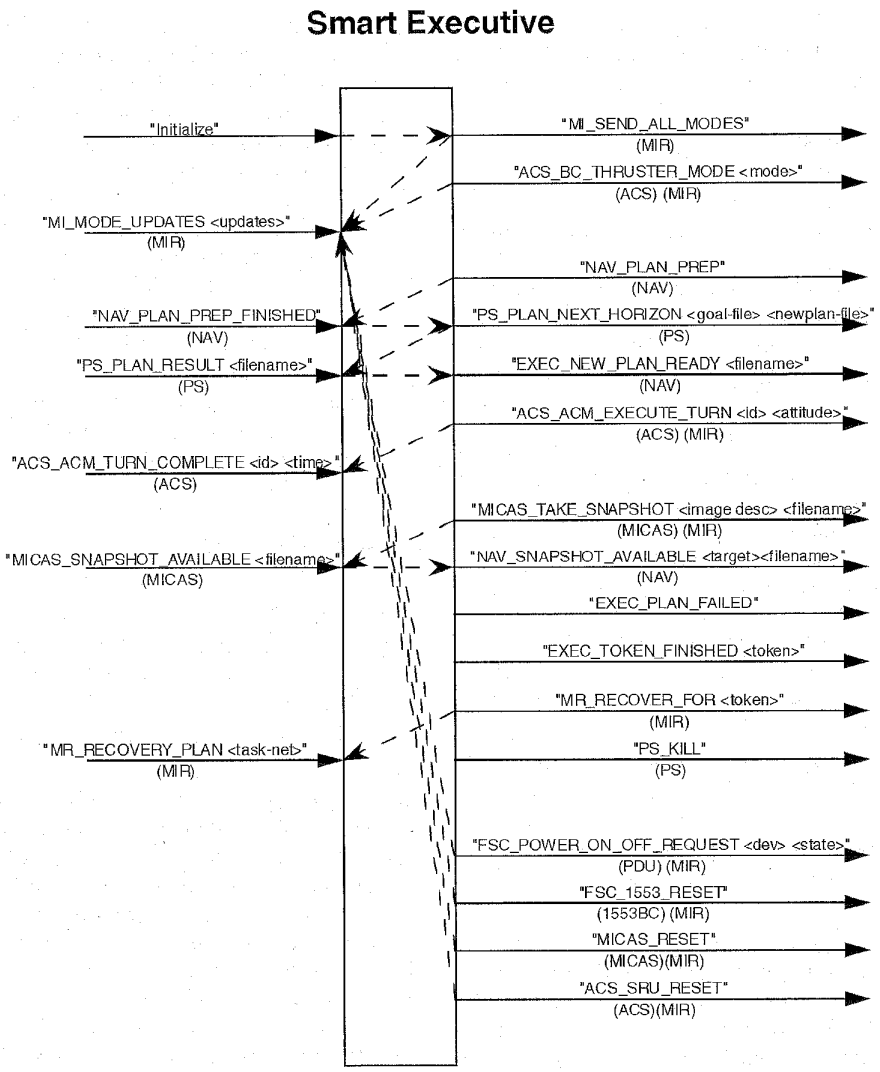


Fig. 1. Diagram of a software module interface.

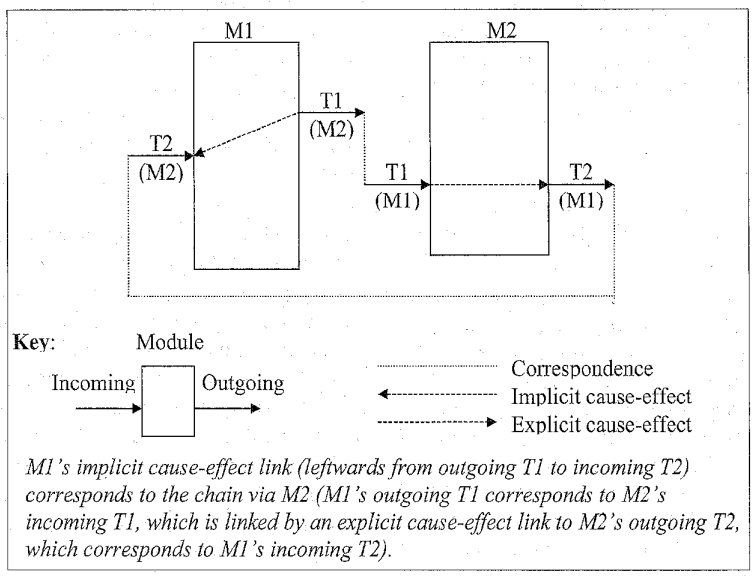


Fig. 2. An implicit cause-effect link and a corresponding chain of explicit cause-effect links.

3.3.1 Representation

The first analysis step was the design of a database representation to hold the information content of the software interface diagrams. The AP5 database provides entities (typed objects) and relationships among entities as building blocks, so a straightforward representation was developed in terms of these, as follows:

Each message arrow was represented as an entity of type arrow, with attributes (binary relations) to hold the arrow's message name, the arrow's destinations or sources, and its parameters. Subtypes of this arrow type distinguished incoming and outgoing arrows. Cause-effect links were represented similarly.

Definitions of information derived from the above were then added. For example, the concept of correspondence between an outgoing arrow and an incoming arrow was defined to hold whenever two such arrows share the same message name, the outgoing arrow points to (i.e., has as one of its destinations) the incoming arrow's module, and the incoming arrow points from (i.e., has as one of its sources) the outgoing arrow's module. The AP5 definition of this "correspondence" concept is shown Fig. 3.

The innermost clauses are relational queries, e.g., (arrow-name oa name) is true if and only if the arrow-name relation relates oa to name. Traditional logical connectives (e.g., and), and quantification (e.g., (E (v) q)—meaning the existential predicate whose bound variable is v and whose inner predicate is q) can be employed.

The syntax combines a lisp heritage and a relational flavor, which to those unfamiliar with either is somewhat obscure at first reading. The important point is to observe that the overall form of the definition mirrors closely the English description above, that is, there is a straightforward rendering of the obvious definition into the corresponding formal expression.

3.3.2 Data Entry

The diagrams were available electronically as PostScript files. Some manipulation was required to extract the information contents of those files and massage it into a form that could then be input into the AP5 database. This was done semi-automatically, mostly by means of Emacs macros to extract the textual contents (message names, parameters, and input/output module names). The information imparted by the cause-effect arrows had to be manually entered into the database (since it was not readily apparent how to recognize from the PostScript file which message types were being connected by a cause-effect arrow). Overall, this process took on the order of a couple of hours to process the equivalent of six times the information content of Fig. 1.

3.3.3 Analysis

The analysis objectives of Section 3.2 were expressed as database queries to evaluate against the data entered in the previous step. The definitions follow:

- *"Dangling" outgoing message types.*

  ```
  (listof (oa) s.t. (and (out-arrow oa)
                         (not (E (ia) (correspondence oa ia)))))
  ```

  retrieves the list of objects that are outgoing arrows and are not related by correspondence to anything.

- *"Dangling" incoming message types (analogous to the previous bullet).*

  ```
  (listof (ia) s.t. (and (in-arrow ia)
                         (not (E (oa) (correspondence oa ia)))))
  ```

- *"Mismatched" parameters.*

  ```
  (listof (oa ia) s.t.
      (and (correspondence oa ia)
           (not (E (plist) (and (parameters oa plist)
                                (parameters ia plist))))))
  ```

  retrieves the list of pairs of objects that are related by correspondence (i.e., an outgoing message and its incoming counterpart), but whose parameter lists are *not* equal.

- *"Miraculous" implicit cause-effect links.*

  ```
  (listof (oa ia) s.t. (and (implicit-cause-effect oa ia)
                            (not (cause-effect-chain oa ia))))
  ```

  retrieves the list of pairs of objects that are related by implicit-cause-effect but *not* related by cause-effect-chain. The definitions of those relations are as follows:

  ```
  (defrelation implicit-cause-effect
  ((oa ia) s.t. (and (out-arrow oa) (in-arrow ia) (cause-effect oa ia))))
  ```

  i.e., a cause-effect arrow from an outgoing message arrow to an incoming one. explicit-cause-effect is defined analogously.

```
(defrelation correspondence          ; define the concept correspondence; it relates
 ((oa ia) s.t.                       ; a pair of objects, oa and ia, such that:
  (and (out-arrow oa)                ; oa is an outgoing message arrow,
       (in-arrow ia)                 ; ia is an incoming message arrow,
       (E (name) (and (arrow-name oa name) (arrow-name ia name)))
                                     ; oa's name and ia's name are the same,
       (E (mod) (and (arrow-module ia mod) (arrow-to oa mod)))
                                     ; oa points to ia's module, and
       (E (mod) (and (arrow-module oa mod) (arrow-from ia mod)))
                                     ; ia points from oa's module.
)))
```

Fig. 3. Example AP5 definition.

```
(defrelation cause-effect-chain
(tclosure correspondence-or-explicit-cause-effect))
```

i.e., the *transitive closure* of correspondence-or-explicit-cause-effect, which is in turn defined by:

```
(defrelation correspondence-or-explicit-cause-effect
    ((a1 a2) s.t. (or (correspondence a1 a2)
                      (explicit-cause-effect a1 a2))))
```

- *"Omitted" implicit cause-effect link.*

```
(listof (oa ia) s.t. (and (not (implicit-cause-effect oa ia))
                          (cause-effect-chain oa ia)))
```

retrieves the list of pairs of objects that are *not* related by implicit-cause-effect but *are* related by cause-effect-chain.

Again, observe that the formal expression of these queries is a straightforward rendering of the informally expressed concepts.

3.4 Analysis Results

The performed analyses resulted in a list of anomalies for each of the analysis objectives. Anomalies were readily categorizable into either:

- *Inconsistency*. Attributable to contradictory information present within the set of software interface diagrams, or
- *Incompleteness*. Attributable to information missing from set of software interface diagrams.

It was immediately clear that a good number of the incompleteness anomalies could be attributed to the fact that diagrams had not been drawn of *all* of the system's software modules. In response to this, a simple refinement of the database queries was made to draw the distinction between the following two subcases of incompleteness:

- *Internal Incompleteness*. Attributable to missing information that should have been present within the existing set of software interface diagrams, and
- *External Incompleteness*. Attributable to the lack of a software interface diagram; these anomalies point to expectations on the information that those diagrams, if provided, would contain.

Finally, one further subcase of reported anomalies was distinguished—"missing" implicit cause-effect links that could be deduced from other links all present in the *same* diagram. For example, for the module shown in, analysis reported that an implicit cause-effect link from the outgoing NAV_PLAN_PREP arrow to the incoming PS_PLAN_RESULT arrow was missing. Upon inspection, it was obvious that the diagrams were not bothering to show such links when they could be deduced from the presence of the other cause-effect links shown within that same diagram (NAV_PLAN_PREP to NAV_PLAN_PREP_FINISHED to PS_PLAN_NEXT_HORIZON to PS_PLAN_RESULT). Again, a simple refinement of the database query was sufficient to automatically distinguish such cases.

What remained was a list of approximately 20 genuine anomalies, distributed over the various categories of anomalies identified above. Note that these anomalies were derived from the information in the software module interface diagrams, which were used as a form of documentation aid for understanding. Hence it should not be construed that these anomalies carried through into the actual design.

3.5 Analysis Discussion

Database as a formal analysis tool. The effort succeeded in straightforwardly representing simple design information in a database, and conducting consistency and completeness checks by issuing queries against that database.

Flexibility and customizability. Flexibility to construct new checks, and to refine previous checks, was crucial. This study employed both generic checks (e.g., correspondences between outgoing and incoming messages) and customized checks (e.g., concerning cause-effect links). Thus there was no a priori limitation on the set of checks that could be conducted. Furthermore, refinement of checks was also found to be useful. In this study, the originally designed consistency checks were refined to take into account whether information was expected to be absent (e.g., reference to information in a diagram not provided; implicit cause-effect links justified by other links present within the same diagram). Such iteration is common whenever someone other than the authors of the design information conducts the analysis. Assumptions on the use of a notation may not have been documented, or the analyst may not have come across such documentation. Analysis that does not take these assumptions into account generates false alarms (typically, a large number of them!) on the first attempt at analysis. The analyst must recognize the likely underlying cause, confirm it with the developers, and refine the analysis accordingly. Flexibility is the key to being able to do this.

Working with available information. Adapting the analysis activities to the existing available information sources was relatively straightforward. While not completely automated, actual data capture was a relatively short process. It was preparing for data capture (i.e., determining the appropriate database schema representation) and conducting the analyses (i.e., expressing them as database queries) that took the bulk of the time and effort.

It is likely that the use of an existing CASE tool would provide for most of the checks that we performed, notably as checks of data flow diagrams. However, one of the goals of this study was to work with whatever form of information was currently available. The developers had not employed a CASE tool to design this portion of the software. Our approach makes it possible to perform the equivalent analyses at little cost, *and* provides for additional (and useful) flexibility to build customized analysis checks. In particular, if we were to try to perform our checks with a CASE tool, we would have had to convert the available information in order to input it into the tool. Additionally, we would likely not have had sufficient flexibility to define additional checks (notably of the implicit cause-effect links) not already provided for by that CASE tool, or to modify checks to account for missing "obvious" information. The freedom of users to invent and employ their own notational variants is obviously desirable. Our study suggests it may be possible to allow this freedom, and be able to rapidly construct analysis mechanisms that work directly with the users' new notations.

It could be argued that the redundancy present in the design information was unfortunate, and that a better solution would have been to use a specification style that eschewed such redundancy. In particular, the implicit causality links could have been deduced automatically from the other cause-and-effect information, and so should never need to have been drawn by hand. Again, we appeal to the goal to work with available information, and stress that we analyzed what we were actually provided with. Additionally, it is plausible that the implicit causality links represented requirements, and that the rest of the specification was intended to fulfill them. If this was indeed the case, then the analysis was useful as a means to check consistency of the requirements against the system description.

4 Pilot Study II—Analyzing Test Logs

In the second pilot study, we sought to repeat the use of the database tool as a mechanism for lightweight, rapid analysis, but applied to a different aspect of the development process. Our second area of focus was on the testing activities of the process.

As part of testing, the software modules are executed in a simulation test-bed (simulating both the hardware, and the hardware's environment, e.g., the spacecraft's camera's view of stars). Each test run results yields transcripts of the software's behavior. In particular, the message passing between the software modules is recorded in log files. The test team studies these logs to check that the software is correctly commanding the spacecraft. However, these logs are highly detailed, and often quite lengthy (several thousand messages were typical for a test run even during the first iteration of the project's development cycle). Hence, it was thought appropriate to perform a pilot study in which rapid lightweight analysis would be applied to a small subset of these test logs.

Again, the four requirements for rapid analysis were considered, and led to the following decisions:

1) *To rapidly acquire the information to be analyzed.* The available test logs provided the raw data for analysis, already in a highly structured and therefore readily machine manipulable form.
2) *To rapidly decide what to analyze this information for.* There are two sources of properties that should be true of spacecraft control, and therefore are immediate candidates for log file analysis: requirements on the correct behavior of the spacecraft itself (e.g., that the camera shall never be pointed too close to the sun), and expected protocols of message flow between the software modules. The latter are important because deviations from expectations might point to abnormalities in the control software itself.
3) *To rapidly perform the analysis itself.* Again, a database was to serve as the analysis engine. The expectation was that it would be straightforward to populate the database with the messages in a log file, and express the requirements to be checked as queries. The query mechanism of the database itself would perform the analyses rapidly and automatically.
4) *To rapidly interpret the results of the method.* Any discrepancies that this analysis revealed would be readily traceable to the problematic message(s) in the log files. Hence, this step was expected to need little further effort.

4.1 Available Information

A test run of the spacecraft software yields (among other things) a log file recording all of the message passing between the software modules. Fig. 4 shows a fragment from such a recording. Each log file line corresponds to a message being sent between modules of the software implementation. For readability here, log file lines longer than the width of this text have been split into multiple lines with their continuations indented.

Each log file line begins with the message name, followed by the arguments to that message. Typically, these arguments take the form of a { } delimited list of { } delimited attribute-value pairs. For example, the first line comprises the message name FSC_POWER_ON_OFF_REQUEST, and a list of two attribute-value pairs, the first of attribute switch_name and value ACS_EGA_ASW1, and the second of attribute switch_state and value FSC_SWITCH_ON. For a few message types, a single argument value simply appears after the message name without { } delimiters. For example, the ACS_MDC_STATE_COMMAND message towards the end of the fragment has argument ACS_RCSDV_MODE.

```
FSC_POWER_ON_OFF_REQUEST {{switch_name ACS_EGA_A_SW1} {switch_state
        FSC_SWITCH_ON}}
MI_MODE_UPDATES {{updates "(MODE-UPDATES (POWER-STATE EGA_A ON)
        ((POWER-STATE EGA_A ON)))"}}
FSC_IPS_SET_THRUST_LEVEL {{level 10}}
MI_MODE_UPDATES {{updates "(MODE-UPDATES (OP-STATE IPS_A (STEADY-STATE
        10)) ((OP-STATE IPS_A STARTUP)))"}}
MI_MODE_UPDATES {{updates "(MODE-UPDATES NIL ((OP-STATE IPS_A (STEADY-
        STATE 10))))"}}
ACS_MDC_STATE_COMMAND ACS_RCSDV_MODE
MI_MODE_UPDATES {{updates "(MODE-UPDATES (CONTROL-MODE ACS
        ACS_TVC_MODE) ((CONTROL-MODE ACS ACS_TVC_MODE)))"}}
```

Fig. 4. Fragment of a log file.

4.2 Analysis Objective

The log files capture message passing between the software modules during test runs. Two kinds of properties can be analyzed for in these log files:

- Whether the controlled spacecraft adheres to all its explicit requirements. For example, that *the boresight of the camera shall never point to within 1 degree of the sun when the camera cover is open.* These requirements are called "flight rules," of which there are expected to be on the order of 100 as successive design iterations introduce the DS-1 spacecraft's complete functionality.
- Whether the control software itself is operating normally. Specifically, whether the message flow between the software modules follows the expected protocols. For example, that a command message is followed some time later by the corresponding confirmation message, before the next such command message is sent. Deviations from these expected protocols might indicate abnormalities in the control software itself.

4.3 Analysis Process

The same database as had been used for the first phase of this pilot study was used again as the analysis tool. Analyses were achieved by:

1) preparing a database representation to hold the information content of the message log files,
2) loading the message log file information into the database,
3) issuing the appropriate database queries corresponding to the properties to be checked of the log file, and
4) interpreting the results.

4.3.1 Representation

The first analysis step was the design of a database representation to hold the information content of the log files. We made the initial decision to load all the messages into the database, and then conduct the analyses as queries against this database.

Messages were represented as objects in the database, with their names and attribute-value pairs represented as attributes of those message objects. Asserting the binary relation msg-then-msg between the objects representing successive messages captured the sequencing of messages in the log file. The transitive closure of this relation was defined as then, allowing the querying of whether two messages appear in a given order in the log file (but are not necessarily immediate successors).

4.3.2 Data Entry

The well-structured form of the log files made the task of parsing them into the database a straightforward programming task.

Some of the log files were quite large–3,000 or more messages were common, in even in the early stages of development. This rendered our naïve approach, loading *all* of a log file's messages into the database, rather slow. Since each query typically involved only a small subset of the message types, it proved to be much more efficient to load into the database only messages that were instances of those message types. This was easily automated, by looking for the message names mentioned in the query, and then loading those and only those named messages. This simple refinement considerably speeded the loading and analysis activities. Discarding information that is obviously irrelevant to the analysis objective is a commonly applied step to improve the tractability of analysis.

4.3.3 Analysis

Analysis was to check adherence to flight rules (explicit requirements expressed in terms of the spacecraft's condition) and conformance to normal operation of the control software itself (protocols of message flow among the software modules).

Analysis was achieved by expressing a flight rule or message protocol as a database query (or queries) which would retrieve all instances of messages in violation of that condition. For example, one of the flight rules says: *"When the IPS is not thrusting, be in RCS control mode."* This could be violated either by turning off thrusting while not in RCS control mode, or turning off RCS control mode while not thrusting. The following two database queries check for these violations:

- Find an IPS_thrust_off message that occurs while RCS control mode is off (i.e., occurs *after* some earlier RCS_control_mode_off message, but *before* any subsequent RCS_ control_mode_on message), and
- Find an RCS_control_mode_off message that occurs while IPS is not thrusting (i.e., occurs *after* some earlier IPS_thrust_off message, but *before* any subsequent IPS_thrust_on message).

It is straightforward to express queries such as these in the AP5 database query language. For example, the query corresponding to the first kind of violation is shown in Fig. 5.

Ideally, the user should be able to express the single flight rule, and have this pair of queries be automatically generated. While we anticipate this would be simple to implement, we did not do so as part of the pilot study.

Analysis for conformance to message passing protocols is similar in nature. For example, we may expect conformance to the protocol that a command message is followed some time later by the corresponding confirmation message, before the next such command message is sent. The database query to check for violations of this rule need simply look for two occurrences of the command message without any intervening confirmation message.

Since there were multiple log files (corresponding to distinct test runs), we found it convenient to automate conducting the same analysis across a whole set of log files. The results for each log file were gathered, together with a summary pointing to which (if any) of the log files yielded nonempty lists of answers to the queries. Again, a simple step to take, but one that further enhanced the automation of the overall process. We found that the time to perform a typical query across a set of 60 or so log files, of which at least 20 were of substantial length, was on the order of 2 min, executing on a Pentium® 166.

```
(listof (m1 m2) s.t.
  (and (RCS_control_mode_off m1)          ; m1 is an RCS_control_mode_off message
       (IPS_thrust_off m2)                ; m2 is an IPS_thrust_off message
       (then m1 m2)                       ; m2 occurs after m1
       (not (exists (m3)                  ; there's no m3 that's an:
            (and (RCS_control_mode_on m3)     ; RCS_control_mode_on message,
                 (then m1 m3)                 ; occurs after m1, and
                 (then m3 m2)))))))           ; occurs before m2
```

Query to find a message that turns off IPS thrusting while RCS is off (i.e., after an RCS off message, but before any subsequent RCS on message).

Fig. 5. Database query for flight rule violations.

4.4 Analysis Results

Analyses were conducted for adherence to a small number of the expected message patterns and flight rules, on log files produced in two successive rounds of the project's iterative development process.

One surprise was revealed by these analyses—while checking a log file for adherence to a flight rule, several violations were detected. This led to (manual) inspection of the log file. Guided by the violation (i.e., knowing what to look for), it was now easy to spot repeated instances where commanding the spacecraft to change to an RCS mode was followed by a confirmation reporting the spacecraft had changed to a mode *other* than RCS! The explanation turned out to be attributable to an anomaly in the creation of the message log files—the actual logging uses concise numerical codes in place the human readable message names. Postprocessing of the log files is performed later to replace the codes with the corresponding names. In between the time of the simulation run (during which the logging of messages actually took place) and the postprocessing of those log files, the correspondence between message names and codes was changed. Thus it turned out to be a false alarm—the spacecraft software had, in fact, been working correctly.

In fact, no genuine anomalies were discovered by this pilot study, most likely because for the patterns and flight rules studied, the design was functioning correctly. Nevertheless, the goals of the pilot study were met—namely, to demonstrate the feasibility of rapid analysis of voluminous amounts of information. This has led to more recent work (ongoing) in which project money (as contrasted to the research funding that supported the pilot study) is paying for the application of this same overall approach.

4.5 Analysis Discussion

Database as a formal analysis tool. Again, a database proved sufficient for representing the information content to be analyzed, and its query mechanism capable of expressing the analyses. The test logs emerge from test runs of the complex software system, which involves a planner, and concurrently operating diagnostic engine and real-time executive. Thorough analysis of these test logs is helpful towards developing assurance that the system as a whole is operating correctly.

Scaling to larger analyses. These experiments were conducted in the early iterations of the spacecraft software development. The simple optimization of loading the database with only those messages relevant to the query, discarding the rest, was sufficient to achieve desirable levels of efficiency for these early iterations. In successive design iteration, further detail of the spacecraft is introduced, leading to more message types, more flight rules, and longer message logs. This might necessitate further optimization of the checking process. The obvious next step to optimize the analysis technique would be to process each message log file incrementally, maintaining just enough of the state of the spacecraft so as to be able to check the flight rule(s) of interest at the time. For example, to check the flight rule: *"When the IPS is not thrusting, be in RCS control mode,"* read the messages of a log file in one by one, maintaining the IPS state (thrusting or not thrusting) and the control mode (RCS or not RCS) incrementally, and watch for a database transition that leads to a state in violation of this rule. This is an example of the conversion of a temporal formula into a finite state automaton to recognize violations of that formula. Such an approach is described in [4]. If we were to incorporate this optimization into our approach, we would encode the automaton as database integrity conditions that watch each database transition, advancing the state of the automaton as appropriate and generating an error report if and when the automaton ever reached a state corresponding to a violation.

Convenience. The experience of expressing several flight rules and message passing protocols revealed commonly recurring idioms. An example of such an idiom is: *the occurrence of an instance of message A followed some time later by an instance of message B without an intervening instance of message C.* Rather than have to write:

```
(exists (mA mB) (and (A_type_message mA)
                     (B_type_message mB)
                     (then mA mB)
                     (not (exists (mC) (and (C_type_message mC)
                                            (then mA mC)
                                            (then mC mB))))))
```

it was much more convenient to define a macro A_noC_B that would expand to the above, and thereafter write simply:

```
(A_noC_B A_type_message B_type_message C_type_message).
```

Indeed, what was needed was a combination of a vocabulary tailored for intervals [1] (e.g., interval A should *contain* interval B), timing constraints [10] (e.g., X must occur within 10 sec after a Y has occurred), etc.

5 Conclusions

The two pilot studies showed two successful applications of lightweight formal methods in a fast-paced development setting, yielding results of a modest nature in a timely fashion. These studies employed lightweight formal methods as a complement to, not replacement for, other forms of quality assurance. Their relationship to other formal method approaches and to testing is discussed next.

Thorough testing of the software code itself remains essential, since it alone has the capacity to exercise the actual code rather than an abstraction of that code. Testing would, of course, reveal the interface mismatches that the first pilot study addresses, however would do so only after the designs had been developed into code. The first pilot study showed how this class of problems could be detected at design time. The inclusion of a form of simple causality information in the design documents made possible further checking, beyond simply issues of interface compatibility. The second pilot study showed how the information that results from testing could be automatically analyzed for a range of conditions. This is obviously reliant upon testing having taken place (without testing, there would be no test logs to analyze!). It augments testing by performing a series of mundane checks carried out thoroughly against all the available test logs. It could also be useful as a means to query the set of test cases, and thus help the test team to navigate through the set of tests, and to estimate the coverage provided by those tests.

More traditional "heavyweight" forms of formal methods, for example, symbolic evaluation of a formal specification, state exploration, and theorem proving, are irreplaceable as means to expose the presence of particularly subtle errors (better yet, to provide assurance as to the absence of those errors). However, these methods typically do not scale to the size and complexity of the full problem. Therefore they are usually applied to a carefully chosen subset of the overall problem, or to a carefully constructed abstraction of the problem. In contrast, the lightweight approach followed here is applied to the information content of existing project documentation, and little or no human-conducted selection from, or abstraction of, this information has been needed. We therefore ascribe the benefits of the approach to the mechanized technique, rather than to the skill and insight of the human analyst. This gives us confidence that the technique is ready for widespread application.

Many approaches to analysis would be *capable* of performing the simple analyses that we have conducted. However, our emphasis has been upon the goal of *rapid* analysis. This precludes many of the heavyweight formal methods, since their application requires considerable time and effort to prepare the input for the analysis tool (e.g., to construct a formal specification), and, often, to perform the actual analysis (e.g., attempt to prove the theorem, and interpret the failure to do so as an anomaly in the specification). An interesting observation (due to one of this paper's reviewers) is that heavyweight formal methods typically require manual effort that is *proportional to the entire input to the analysis process,* including both the properties to be shown *and* the specification upon which that analysis is to be conducted. For example, theorem proving usually requires manual effort to construct the specification, to express the properties to be proven, and then to actually prove them. Model checking and other forms of state exploration require construction of a state machine model to be checked; often, to obtain a state model that is tractable to analyze, some significant amount of manually performed abstraction is required. The lightweight approach as applied in the second pilot study required manual effort that was proportional only to the expression of the properties—the test logs to be checked for adherence to these properties were loaded automatically into the database. This made it suited to the checking of voluminous amounts of information. The first pilot study did require some manual effort proportional to the size of the specifications—it was necessary to manually translate the information content of diagrams' arrows into database information. However, this was so straightforward a task that it could be done fairly quickly and without great insight.

Overall, we attribute the achievement of rapid analysis to our choice of a database as the analysis engine. This enabled us to quickly and easily incorporate available information as data in the database, and cast analysis problems as database queries. The underlying database technology then provided the efficient evaluation of those queries.

Nevertheless, there *are* emerging results of application of "heavyweight" formal methods in a rapid enough fashion to contribute to ongoing development activities. For example, [5] presents three case studies of formal methods applied to the requirements phase of system developments. In these cases there is typically an up-front manual activity of reexpressing the requirements into a form suitable for application of the analysis tools. This manual activity yields benefits even before the analysis takes place (e.g., revelations of minor ambiguities and incompletenesses), so itself qualifies as a form of "lightweight" analysis method, with the added benefit of being a key step towards the eventual application of the "heavyweight" method. Further research is needed find a smooth progression from "lightweight" to "heavyweight" formal methods, and so realize the benefits of both approaches. Work along these lines includes the ongoing extension of SCR-style consistency checking (e.g., [7]) to incorporate model checking as a more sophisticated form of analysis, and, in the reverse direction, the incorporation of tabular forms of expression into the PVS theorem prover [12].

The most expedient applications of formal methods occur when the form of input needed by the formal analysis tool coincides with the form of expression used by the project to state their requirements, designs, etc. CASE tools would seem to offer an expedient vehicle through which to both capture the input needed for formal analysis, and provide (automatic) formal analysis as just another option to the user. The relatively simple kinds of analyses reported herein might be well suited to CASE tool usage. More sophisticated analyses, however, might necessitate some tailoring of the input language. For example, [9] devised a formal language that was suitable for human writing and reviewing of requirements *and* for mechanical analysis. Such circumstances are all too rare, and it is likely that for a long time to come there will remain a need to adapt formal

method techniques to whatever forms of documentation are employed by projects. Lightweight formal methods seem particularly suited to this activity.

Another possible application of lightweight formal methods is as part of the infrastructure that would support "viewpoints" (also called "multiple perspective") during software development [6], [13]. In this envisaged approach, a development environment should actively support multiple participants in the course of their requirements, design, etc., activities. Consistency checking between the multiple participants' artifacts (e.g., requirements, specifications, and designs) would be a core service of any such environment.

ACKNOWLEDGMENTS

Thanks to John Kelly (within whose formal methods effort this work has been conducted) and Robyn Lutz for insights and support. The assistance of many New Millenium DS-1 project members is gratefully acknowledged, especially Bob Kanefsky from NASA/AMES and Tom Starbird from NASA/JPL, who have answered many questions and provided many clarifications. The generosity of the New Millenium teams to make their interim project development documentation available to us for study has made this project possible. The editors' and referees' comments were insightful and helpful.

The research described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Funding was provided under NASA's Code Q Software Program Center Initiative UPN #323-08. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States government or the Jet Propulsion Laboratory, California Institute of Technology.

REFERENCES

[1] J.F. Allen, "Maintaining Knowledge about Temporal Intervals," *Comm. ACM*, vol. 26, no. 11, pp. 832-843, 1983.

[2] B.W. Boehm, *Software Engineering Economics*, Englewood Cliffs, N.J.: Prentice Hall, 1981.

[3] D. Cohen, "Compiling Complex Database Transition Triggersm," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 225-234, Portland, Ore., ACM Press, 1989.

[4] L. Dillon and Q. Yu, "Oracles for Checking Temporal Properties of Concurrent Systems," *Proc. Second ACM SIGSOFT Symp. Foundations of Software Eng.*, Software Engineering Notes, vol. 19, no. 5, pp. 140-153, 1994.

[5] S. Easterbrook, R. Lutz, R. Covington, J. Kelly, Y. Ampo, and D. Hamilton, "Experiences Using Formal Methods for Requirements Modeling," *IEEE Trans. Software Eng.*, vol. 24, no. 1, Jan. 1998.

[6] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke, "Viewpoints: A Framework for Integrating Multiple Perspectives in System Development," *Int'l J. Software Eng. and Knowledge Eng.*, vol. 2, no. 1, pp. 31-58, Mar. 1992.

[7] C. Heitmeyer, R. Jeffords, and B. Labaw, "Automated Consistency Checking of Requirements Specifications," *ACM Trans. Software Eng. and Methodology*, vol. 5, no. 3, pp. 231-261, July 1996.

[8] D. Jackson and J. Wing, "Lightweight Formal Methods," *Computer*, pp. 21-22, Apr. 1996.

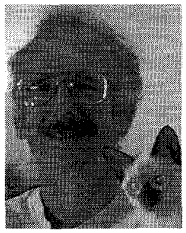
[9] N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, and J.D. Reese, "Requirements Specification for Process-Control Systems," *IEEE Trans. Software Eng.*, vol. 20, no. 9, pp. 684-707, Sept. 1994.

[10] R. Lutz and J. Wong, "Detecting Unsafe Error Recovery Schedules," *IEEE Trans. Software Eng.*, vol. 18, no. 8, pp. 749-760, 1992.

[11] R. Lutz, "Analyzing Software Requirements Errors in Safety-Critical, Embedded Systems," *Proc. IEEE Int'l Symp. Requirements Eng., RE '93*, pp. 126-133, San Diego, Calif., Jan. 1993.

[12] S. Owry, J. Rushby, and N. Shankar, "Integration in PVS: Tables, Types, and Model Checking," *Tools and Algorithms for the Construction and Analysis of Systems, TACAS'97*, Enschede: The Netherlands, 1997.

[13] *Joint Proc. SIGSOFT'96 Workshops, Part II: Proc. Int'l Workshop Multiple Perspectives in Software Development*, L. Vidal, A. Finkelstein, G. Spanoudakis, and A. Wolf, eds., ACM 1996.



Martin S. Feather obtained his BA and MA degrees in mathematics and computer science from Cambridge University, England, and his PhD degree in artificial intelligence from the University of Edinburgh, Scotland. For a number of years, Dr. Feather worked on National Science Foundation and Defense Advanced Research Projects Agency-funded research while at the University of Southern California's Information Sciences Institute. Currently, he is a member of the technical staff at the Jet Propulsion Laboratory, California Institute of Technology, Pasadena, and a research scientist at Computing Support Services Solutions, Los Angeles. His research interests encompass the early phases of software development, especially analysis and implementation of requirements and specifications.